Python Cookbook

Recipes for Mastering Python 3

第3版

Python Cookbook™

中文版

[美] *David Beazley & Brian K. Jones* 著

陈舸 译





O'REILLY®

Python Cookbook
（第3版）中文版

[美] David Beazley　Brian K.Jones　著

陈　舸　译

人 民 邮 电 出 版 社

北　京

版权声明

◆ 著 [美] David Beazley Brian K.Jones
译 陈 舸
责任编辑 傅道坤
责任印制 张佳莹
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷有限公司印刷
◆ 开本：787×1000 1/16
印张：43.75
字数：914 千字 2015 年 2 月第 1 版
印数：1–0 000 册 2015 年 2 月北京第 1 次印刷
著作权合同登记号 图字：01-2013-7656 号
定价：00.00 元
读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书介绍了 Python 应用在各个领域中的一些使用技巧和方法，其主题涵盖了数据结构和算法，字符串和文本，数字、日期和时间，迭代器和生成器，文件和 I/O，数据编码与处理，函数，类与对象，元编程，模块和包，网络和 Web 编程，并发，实用脚本和系统管理，测试、调试以及异常，C 语言扩展等。

本书覆盖了 Python 应用中的很多常见问题，并提出了通用的解决方案。书中包含了大量实用的编程技巧和示例代码，并在 Python 3.3 环境下进行了测试，可以很方便地应用到实际项目中去。此外，本书还详细讲解了解决方案是如何工作的，以及为什么能够工作。

本书非常适合具有一定编程基础的 Python 程序员阅读参考。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的"动物书"；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

"O'Reilly Radar 博客有口皆碑。"

——Wired

"O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。"

——Business 2.0

"O'Reilly Conference 是聚集关键思想领袖的绝对典范。"

——CRN

"一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。"

——Irish Times

"Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：'如果你在路上遇到岔路口，走小路（岔路）。'回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。"

——Linux Journal

前言

自 2008 年以来，我们已经目睹了整个 Python 世界正缓慢向着 Python 3 进化的事实。众所周知，完全接纳 Python 3 要花很长的时间。事实上，就在写作本书时（2013 年），大多数 Python 程序员仍然坚持在生产环境中使用 Python 2。关于 Python 3 不能向后兼容的事实也已经做了许多努力来补救。的确，向后兼容性对于任何已经存在的代码库来说是个问题。但是，如果你着眼于未来，你会发现 Python 3 带来的好处绝非那么简单。

正因为 Python 3 是着眼于未来的，本书在之前的版本上做了很大程度的修改。首先也是最重要的一点，这是一本积极拥抱 Python 3 的书。所有的章节都采用 Python 3.3 来编写并进行了验证，没有考虑老的 Python 版本或者“老式”的实现方式。事实上，许多章节都只适用于 Python 3.3 甚至更高的版本。这么做可能会有风险，但是最终的目的是要编写一本 Python 3 的秘籍，尽可能基于最先进的工具和惯用法。我们希望本书可以指导人们用 Python 3 编写新的代码，或者帮助开发人员将已有的代码升级到 Python 3。

无需赘言，以这种风格来编写本书给编辑工作带来了一定的挑战。只要在网络上搜索一下 Python 秘籍，立刻就能在 ActiveState 的 Python 版块或者 Stack Overflow 这样的站点上找到数以千计的使用心得和秘籍。但是，大部分这类资源已经沉浸在历史和过去中了。由于这些心得和秘籍几乎完全是针对 Python 2 所写的，其中常常包含有各种针对 Python 不同版本（例如 2.3 版对比 2.4 版）之间差异的变通方法和技巧。此外，这些网上资源常常使用过时的技术，而这些技术现在成了 Python 3.3 的内建功能。想寻找专门针对 Python 3 的资源会比较困难。

本书并非搜寻特定于 Python 3 方面的秘籍将其汇集而成，本书的主题都是在创作中由现有的代码和技术而产生出的灵感。我们将这些思想作为跳板，尽可能采用最现代化的 Python 编程技术来写作，因此本书的内容完全是原创性的。对于任何希望以现代化的风格来编写代码的人，本书都可以作为参考手册。

在选择应该包含哪些章节时，我们有一个共识。那就是根本不可能编写一本涵盖了每种 Python 用途的书。因此，我们在主题上优先考虑 Python 语言核心方面的内容，以及能够广泛适用于各种应用领域的常见任务。此外，有许多秘籍是用来说明在 Python 3 中新增的功能，这对许多人来说比较陌生，甚至对于那些使用老版 Python 经验丰富的程序员也是如此。我们也会优先选择普遍适用的编程技术（即，编程模式）作为主

题，而不会选择那些试图解决一个非常具体的实际问题但适用范围太窄的内容。尽管在部分章节中也提到了特定的第三方软件包，但本书绝大多数章节都只关注语言核心和标准库。

本书适合谁

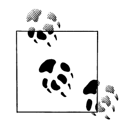
本书的目标读者是希望加深对 Python 语言的理解以及学习现代化编程惯用法的有经验的程序员。本书许多内容把重点放在库、框架和应用中使用的高级技术上。本书假设读者已经有了理解本书主题的必要背景知识（例如对计算机科学的一般性知识、数据结构、复杂度计算、系统编程、并发、C 语言编程等）。此外，本书中提到的秘籍往往只是一个框架，意在提供必要的信息让读者可以起步，但是需要读者自己做更多的研究来填补其中的细节。因此，我们假设读者知道如何使用搜索引擎以及优秀的 Python 在线文档。

有一些更加高级的章节将作为读者耐心阅读的奖励。这些章节对于理解 Python 底层的工作原理提供了深刻的见解。你将学到新的技巧和技术，可以将这些知识运用到自己的代码中去。

本书不适合谁

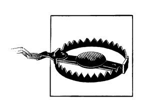
这不是一本用来给初学者首次学习 Python 编程而使用的书。事实上，本书已经假设读者通过 Python 教程或者入门书籍了解了基本知识。本书同样不能用来作为快速参考手册（即，快速查询特定模块中的某个函数）。相反，本书的目标是把重点放在特定的编程主题上，展示可能的解决方案并以此作为跳板引导读者学习更加高级的内容。这些内容你可能会在网上或者参考书中遇到过。

本书中的约定



提示

这个图标用来强调一个提示、建议或一般说明。



警告

这个图标用来说明一个警告或注意事项。

在线代码示例

本书中几乎所有的代码示例都可以在 http://github.com/dabeaz/python-cookbook 上找到。作者欢迎读者针对代码示例提供 bug 修正、改进以及评论。

使用代码示例

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无需获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发 O'Reilly 图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过。一个属性信息通常包括书名、作者、出版社和 ISBN。例如：Python Cookbook, 3rd edtion, by David Beazley and Brain K.Jones(O'Reilly)。Copyright 2013 David Beazley and Brain Jones, 978-1-449-34037-7。

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O'Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过链接 http://oreil.ly/python_cookbook_3e 来访问页面。

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站（http://www.oreilly.com），查看更多我们的书籍、课程、会议和最新动态等信息。

Facebook: http://facebook.com/oreilly

Twitter: http://twitter.com/oreillymedia

YouTube: http://www.youtube.com/oreillymedia

致谢

我们要感谢本书的技术校审人员，他们是 Jake Vanderplas、Robert Kern 以及 Andrea Crotti。感谢他们非常有用的评价，也要感谢整个 Python 社区的支持和鼓励。我们也要感谢本书第 2 版的编辑 Alex Martelli、Anna Ravenscroft 以及 David Ascher。尽管本书的第 3 版是新创作的，但之前的版本为本书提供了挑选主题以及所感兴趣的秘籍的初始框架。最后也是最重要的是，我们要感谢本书早期版本的读者，感谢你们为本书的改进做出的评价和建议。

David Beazley 的致谢

写一本书绝非易事。因此，我要感谢我的妻子 Paula 以及我的两个儿子，感谢你们的耐心以及支持。本书中的许多素材都来自于我过去 6 年里所教的与 Python 相关的训练课程。因此，我要感谢所有参加了我的课程的学生，正是你们最终促成了本书的问世。我也要感谢 Ned Batchelder、Travis Oliphant、Peter Wang、Brain Van de Ven、Hugo Shi、Raymond Hettinger、Michael Foord 以及 Daniel Klein，感谢他们飞到世界各地去教学，而让我可以留在芝加哥的家中完成本书的写作。感谢来自 O'Reilly 的 Meghan Blanchette 以及 Rachel Roumeliotis，你们见证了本书的创作过程，当然也经历了那些无法预料到的延期。最后也是最重要的是，我要感谢 Python 社区不间断的支持，以及容忍我那不着调的胡思乱想。

David M.Beazley

http://www.dabeaz.com

https://twitter.com/dabeaz

Brain Jones 的致谢

我要感谢我的合著者 David Beazley 以及 O'Reilly 的 Meghan Blanchette 和 Rachel Roumeliotis，感谢你们和我一起完成了本书的创作。我也要感谢我的妻子 Natasha，感谢你在我写作本书时给予的耐心和鼓励，也要谢谢你对于我所有追求的支持。我尤其要感谢 Python 社区。虽然我已经在多个开源项目和编程语言中有所贡献，但与 Python 社区长久以来所做的如此令人欣慰和富有意义的工作相比，我做的算不上什么。

Brain K.Jones

http://www.protocolostomy.com

https://twitter.com/bkjones

目录

第 1 章

数据结构和算法

Python 内置了许多非常有用的数据结构，比如列表（list）、集合（set）以及字典（dictionary）。就绝大部分情况而言，我们可以直接使用这些数据结构。但是，通常我们还需要考虑比如搜索、排序、排列以及筛选等这一类常见的问题。因此，本章的目的就是来讨论常见的数据结构和同数据有关的算法。此外，在 collections 模块中也包含了针对各种数据结构的解决方案。

1.1 将序列分解为单独的变量

1.1.1 问题

我们有一个包含 N 个元素的元组或序列，现在想将它分解为 N 个单独的变量。

1.1.2 解决方案

任何序列（或可迭代的对象）都可以通过一个简单的赋值操作来分解为单独的变量。唯一的要求是变量的总数和结构要与序列相吻合。例如：

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
```

```
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

如果元素的数量不匹配，将得到一个错误提示。例如：

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

1.1.3 讨论

实际上不仅仅只是元组或列表，只要对象恰好是可迭代的，那么就可以执行分解操作。这包括字符串、文件、迭代器以及生成器。比如：

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

当做分解操作时，有时候可能想丢弃某些特定的值。Python 并没有提供特殊的语法来实现这一点，但是通常可以选一个用不到的变量名，以此来作为要丢弃的值的名称。例如：

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
```

```
50
>>> price
91.1
>>>
```

但是请确保选择的变量名没有在其他地方用到过。

1.2 从任意长度的可迭代对象中分解元素

1.2.1 问题

需要从某个可迭代对象中分解出 N 个元素，但是这个可迭代对象的长度可能超过 N，这会导致出现“分解的值过多（too many values to unpack）”的异常。

1.2.2 解决方案

Python 的“*表达式”可以用来解决这个问题。例如，假设开设了一门课程，并决定在期末的作业成绩中去掉第一个和最后一个，只对中间剩下的成绩做平均分统计。如果只有 4 个成绩，也许可以简单地将 4 个都分解出来，但是如果有 24 个呢？*表达式使这一切都变得简单：

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

另一个用例是假设有一些用户记录，记录由姓名和电子邮件地址组成，后面跟着任意数量的电话号码。则可以像这样分解记录：

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

不管需要分解出多少个电话号码（甚至没有电话号码），变量 phone_numbers 都一直是列表，而这是毫无意义的。如此一来，对于任何用到了变量 phone_numbers 的代码都不必对它可能不是一个列表的情况负责，或者额外做任何形式的类型检查。

由*修饰的变量也可以位于列表的第一个位置。例如，比方说用一系列的值来代表公司过去 8 个季度的销售额。如果想对最近一个季度的销售额同前 7 个季度的平均值做比

较，可以这么做：

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

从 Python 解释器的角度来看，这个操作是这样的：

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

1.2.3 讨论

对于分解未知或任意长度的可迭代对象，这种扩展的分解操作可谓是量身定做的工具。通常，这类可迭代对象中会有一些已知的组件或模式（例如，元素 1 之后的所有内容都是电话号码），利用*表达式分解可迭代对象使得开发者能够轻松利用这些模式，而不必在可迭代对象中做复杂花哨的操作才能得到相关的元素。

*式的语法在迭代一个变长的元组序列时尤其有用。例如，假设有一个带标记的元组序列：

```
records = [
     ('foo', 1, 2),
      ('bar', 'hello'),
      ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
elif tag == 'bar':
        do_bar(*args)
```

当和某些特定的字符串处理操作相结合，比如做拆分（splitting）操作时，这种*式的语法所支持的分解操作也非常有用。例如：

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
```

```
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

有时候可能想分解出某些值然后丢弃它们。在分解的时候，不能只是指定一个单独的*，但是可以使用几个常用来表示待丢弃值的变量名，比如_或者 ign（ignored）。例如：

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

*分解操作和各种函数式语言中的列表处理功能有着一定的相似性。例如，如果有一个列表，可以像下面这样轻松将其分解为头部和尾部：

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

在编写执行这类拆分功能的函数时，人们可以假设这是为了实现某种精巧的递归算法。例如：

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

但是请注意，递归真的不算是 Python 的强项，这是因为其内在的递归限制所致。因此，最后一个例子在实践中没太大的意义，只不过是一点学术上的好奇罢了。

1.3 保存最后 N 个元素

1.3.1 问题

我们希望在迭代或是其他形式的处理过程中对最后几项记录做一个有限的历史记

录统计。

1.3.2 解决方案

保存有限的历史记录可算是 collections.deque 的完美应用场景了。例如，下面的代码对一系列文本行做简单的文本匹配操作，当发现有匹配时就输出当前的匹配行以及最后检查过的 N 行文本。

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

1.3.3 讨论

如同上面的代码片段中所做的一样，当编写搜索某项记录的代码时，通常会用到含有 yield 关键字的生成器函数。这将处理搜索过程的代码和使用搜索结果的代码成功解耦开来。如果对生成器还不熟悉，请参见 4.3 节。

deque(maxlen=N)创建了一个固定长度的队列。当有新记录加入而队列已满时会自动移除最老的那条记录。例如：

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

尽管可以在列表上手动完成这样的操作（append、del），但队列这种解决方案要优雅得多，运行速度也快得多。

更普遍的是，当需要一个简单的队列结构时，deque 可祝你一臂之力。如果不指定队列的大小，也就得到了一个无界限的队列，可以在两端执行添加和弹出操作，例如：

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

从队列两端添加或弹出元素的复杂度都是 O(1)。这和列表不同，当从列表的头部插入或移除元素时，列表的复杂度为 O(N)。

1.4　找到最大或最小的 N 个元素

1.4.1　问题

我们想在某个集合中找出最大或最小的 N 个元素。

1.4.2　解决方案

heapq 模块中有两个函数——nlargest()和 nsmallest()——它们正是我们所需要的。例如：

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

这两个函数都可以接受一个参数 key，从而允许它们工作在更加复杂的数据结构之上。例如：

```
portfolio = [
   {'name': 'IBM', 'shares': 100, 'price': 91.1},
```

```
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

1.4.3 讨论

如果正在寻找最大或最小的 N 个元素，且同集合中元素的总数目相比，N 很小，那么下面这些函数可以提供更好的性能。这些函数首先会在底层将数据转化成列表，且元素会以堆的顺序排列。例如：

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

堆最重要的特性就是 heap[0]总是最小那个的元素。此外，接下来的元素可依次通过 heapq.heappop()方法轻松找到。该方法会将第一个元素（最小的）弹出，然后以第二小的元素取而代之（这个操作的复杂度是 O(logN)，N 代表堆的大小）。例如，要找到第 3 小的元素，可以这样做：

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

当所要找的元素数量相对较小时，函数 nlargest()和 nsmallest()才是最适用的。如果只是简单地想找到最小或最大的元素（N=1 时），那么用 min()和 max()会更加快。同样，如果 N 和集合本身的大小差不多大，通常更快的方法是先对集合排序，然后做切片操作（例如，使用 sorted(items)[:N]或者 sorted(items)[-N:]）。应该要注意的是，nlargest()和 nsmallest()的实际实现会根据使用它们的方式而有所不同，可能会相应作出一些优化措施（比如，当 N 的大小同输入大小很接近时，就会采用排序的方法）。

使用本节的代码片段并不需要知道如何实现堆数据结构，但这仍然是一个有趣也是值

得去学习的主题。通常在优秀的算法和数据结构相关的书籍里都能找到堆数据结构的实现方法。在 heapq 模块的文档中也讨论了底层实现的细节。

1.5 实现优先级队列

1.5.1 问题

我们想要实现一个队列，它能够以给定的优先级来对元素排序，且每次 pop 操作时都会返回优先级最高的那个元素。

1.5.2 解决方案

下面的类利用 heapq 模块实现了一个简单的优先级队列：

```
import heapq
class PriorityQueue:

    def __init__(self):
        self._queue = []
        self._index = 0

def push(self, item, priority):
    heapq.heappush(self._queue, (-priority, self._index, item))
    self._index += 1

def pop(self):
    return heapq.heappop(self._queue)[-1]
```

下面是如何使用这个类的例子：

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
```

```
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

请注意观察，第一次执行 pop()操作时返回的元素具有最高的优先级。我们也观察到拥有相同优先级的两个元素（foo 和 grok）返回的顺序同它们插入到队列时的顺序相同。

1.5.3 讨论

上面的代码片段的核心在于 heapq 模块的使用。函数 heapq.heappush()以及 heapq.heappop()分别实现将元素从列表_queue 中插入和移除，且保证列表中第一个元素的优先级最低（如 1.4 节所述）。heappop()方法总是返回“最小”的元素，因此这就是让队列能弹出正确元素的关键。此外，由于 push 和 pop 操作的复杂度都是 O(logN)，其中 N 代表堆中元素的数量，因此就算 N 的值很大，这些操作的效率也非常高。

在这段代码中，队列以元组(-priority, index, item)的形式组成。把 priority 取负值是为了让队列能够按元素的优先级从高到低的顺序排列。这和正常的堆排列顺序相反，一般情况下堆是按从小到大的顺序排序的。

变量 index 的作用是为了将具有相同优先级的元素以适当的顺序排列。通过维护一个不断递增的索引，元素将以它们入队列时的顺序来排列。但是，index 在对具有相同优先级的元素间做比较操作时同样扮演了重要的角色。

为了说明 Item 实例是没法进行次序比较的，我们来看下面这个例子：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果以元组(priority, item)的形式来表示元素，那么只要优先级不同，它们就可以进行比较。但是，如果两个元组的优先级值相同，做比较操作时还是会像之前那样失败。例如：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
```

```
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通过引入额外的索引值，以(prioroty, index, item)的方式建立元组，就可以完全避免这个问题。因为没有哪两个元组会有相同的 index 值(一旦比较操作的结果可以确定，Python 就不会再去比较剩下的元组元素了)：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果想将这个队列用于线程间通信，还需要增加适当的锁和信号机制。请参见 12.3 节的示例学习如何去做。

关于堆的理论和实现在 heapq 模块的文档中有着详细的示例和相关讨论。

1.6 在字典中将键映射到多个值上

1.6.1 问题

我们想要一个能将键（key）映射到多个值的字典（即所谓的一键多值字典[multidict]）。

1.6.2 解决方案

字典是一种关联容器，每个键都映射到一个单独的值上。如果想让键映射到多个值，需要将这多个值保存到另一个容器如列表或集合中。例如，可能会像这样创建字典：

```
d = {
   'a' : [1, 2, 3],
   'b' : [4, 5]
}

e = {
   'a' : {1, 2, 3},
   'b' : {4, 5}
}
```

要使用列表还是集合完全取决于应用的意图。如果希望保留元素插入的顺序，就用列

表。如果希望消除重复元素（且不在意它们的顺序），就用集合。

为了能方便地创建这样的字典，可以利用 collections 模块中的 defaultdict 类。defaultdict 的一个特点就是它会自动初始化第一个值，这样只需关注添加元素即可。例如：

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

关于 defaultdict，需要注意的一个地方是，它会自动创建字典表项以待稍后的访问（即使这些表项当前在字典中还没有找到）。如果不想要这个功能，可以在普通的字典上调用 setdefault()方法来取代。例如：

```
d = {} # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...
```

然而，许多程序员觉得使用 setdefault()有点不自然——更别提每次调用它时都会创建一个初始值的新实例了（例子中的空列表[]）。

1.6.3 讨论

原则上，构建一个一键多值字典是很容易的。但是如果试着自己对第一个值做初始化操作，这就会变得很杂乱。例如，可能会写下这样的代码：

```
d = {}
for key, value in pairs:
    if key not in d:
         d[key] = []
    d[key].append(value)
```

使用 defaultdict 后代码会清晰得多：

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

这一节的内容同数据处理中的记录归组问题有很强的关联。请参见 1.15 节的示例。

1.7 让字典保持有序

1.7.1 问题

我们想创建一个字典，同时当对字典做迭代或序列化操作时，也能控制其中元素的顺序。

1.7.2 解决方案

要控制字典中元素的顺序，可以使用 collections 模块中的 OrderedDict 类。当对字典做迭代时，它会严格按照元素初始添加的顺序进行。例如：

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
print(key, d[key])
```

当想构建一个映射结构以便稍后对其做序列化或编码成另一种格式时，OrderedDict 就显得特别有用。例如，如果想在进行 JSON 编码时精确控制各字段的顺序，那么只要首先在 OrderedDict 中构建数据就可以了。

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

1.7.3 讨论

OrderedDict 内部维护了一个双向链表，它会根据元素加入的顺序来排列键的位置。第一个新加入的元素被放置在链表的末尾。接下来对已存在的键做重新赋值不会改变键的顺序。

请注意 OrderedDict 的大小是普通字典的 2 倍多，这是由于它额外创建的链表所致。因此，如果打算构建一个涉及大量 OrderedDict 实例的数据结构（例如从 CSV 文件中读取 100000 行内容到 OrderedDict 列表中），那么需要认真对应用做需求分析，从而判断

使用 OrderedDict 所带来的好处是否能超越因额外的内存开销所带来的缺点。

1.8 与字典有关的计算问题

1.8.1 问题

我们想在字典上对数据执行各式各样的计算（比如求最小值、最大值、排序等）。

1.8.2 解决方案

假设有一个字典在股票名称和对应的价格间做了映射：

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

为了能对字典内容做些有用的计算，通常会利用 zip()将字典的键和值反转过来。例如，下面的代码会告诉我们如何找出价格最低和最高的股票。

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

同样，要对数据排序只要使用 zip()再配合 sorted()就可以了，比如：

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                  (45.23, 'ACME'), (205.55, 'IBM'),
#                 (612.78, 'AAPL')]
```

当进行这些计算时，请注意 zip()创建了一个迭代器，它的内容只能被消费一次。例如下面的代码就是错误的：

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names))    # OK
print(max(prices_and_names))    # ValueError: max() arg is an empty sequence
```

1.8.3 讨论

如果尝试在字典上执行常见的数据操作，将会发现它们只会处理键，而不是值。例如：

```
min(prices)    # Returns 'AAPL'
max(prices)    # Returns 'IBM'
```

这很可能不是我们所期望的，因为实际上我们是尝试对字典的值做计算。可以利用字典的 values()方法来解决这个问题：

```
min(prices.values())    # Returns 10.75
max(prices.values())    # Returns 612.78
```

不幸的是，通常这也不是我们所期望的。比如，我们可能想知道相应的键所关联的信息是什么（例如哪支股票的价格最低？）

如果提供一个 key 参数传递给 min()和 max()，就能得到最大值和最小值所对应的键是什么。例如：

```
min(prices, key=lambda k: prices[k])    # Returns 'FB'
max(prices, key=lambda k: prices[k])    # Returns 'AAPL'
```

但是，要得到最小值的话，还需要额外执行一次查找。例如：

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

利用了 zip()的解决方案是通过将字典的键-值对“反转”为值-键对序列来解决这个问题的。

当在这样的元组上执行比较操作时，值会先进行比较，然后才是键。这完全符合我们的期望，允许我们用一条单独的语句轻松的对字典里的内容做整理和排序。

应该要注意的是，当涉及（value, key）对的比较时，如果碰巧有多个条目拥有相同的 value 值，那么此时 key 将用来作为判定结果的依据。例如，在计算 min()和 max()时，如果碰巧 value 的值相同，则将返回拥有最小或最大 key 值的那个条目。示例如下：

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9 在两个字典中寻找相同点

1.9.1 问题

有两个字典，我们想找出它们中间可能相同的地方（相同的键、相同的值等）。

1.9.2 解决方案

考虑如下两个字典：

```
a = {
   'x' : 1,
   'y' : 2,
   'z' : 3
}

b = {
   'w' : 10,
   'x' : 11,
   'y' : 2
}
```

要找出这两个字典中的相同之处，只需通过 keys()或者 items()方法执行常见的集合操作即可。例如：

```
# Find keys in common
a.keys() & b.keys() # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

这些类型的操作也可用来修改或过滤掉字典中的内容。例如，假设想创建一个新的字典，其中会去掉某些键。下面是使用了字典推导式的代码示例：

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

1.9.3 讨论

字典就是一系列键和值之间的映射集合。字典的 keys()方法会返回 keys-view 对象，其中暴露了所有的键。关于字典的键有一个很少有人知道的特性，那就是它们也支持常见的集合操作，比如求并集、交集和差集。因此，如果需要对字典的键做常见的集合操作，那么就能直接使用 keys-view 对象而不必先将它们转化为集合。

字典的 items()方法返回由(key,value)对组成的 items-view 对象。这个对象支持类似的集合操作，可用来完成找出两个字典间有哪些键值对有相同之处的操作。

尽管类似，但字典的 values()方法并不支持集合操作。部分原因是因为在字典中键和值是不同的，从值的角度来看并不能保证所有的值都是唯一的。单这一条原因就使得某

些特定的集合操作是有问题的。但是，如果必须执行这样的操作，还是可以先将值转化为集合来实现。

1.10 从序列中移除重复项且保持元素间顺序不变

1.10.1 问题

我们想去除序列中出现的重复元素，但仍然保持剩下的元素顺序不变。

1.10.2 解决方案

如果序列中的值是可哈希（hashable）的，那么这个问题可以通过使用集合和生成器轻松解决。示例如下[①]：

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

这里是如何使用这个函数的例子：

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

只有当序列中的元素是可哈希的时候才能这么做。如果想在不可哈希的对象（比如列表）序列中去除重复项，需要对上述代码稍作修改：

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

这里参数 key 的作用是指定一个函数用来将序列中的元素转换为可哈希的类型，这么做的目的是为了检测重复项。它可以像这样工作：

[①] 如果一个对象是可哈希的，那么在它的生存期内必须是不可变的，它需要有一个__hash__()方法。整数、浮点数、字符串、元组都是不可变的。——译者注

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

如果希望在一个较复杂的数据结构中，只根据对象的某个字段或属性来去除重复项，那么后一种解决方案同样能完美工作。

1.10.3　讨论

如果想要做的只是去除重复项，那么通常足够简单的办法就是构建一个集合。例如：

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

但是这种方法不能保证元素间的顺序不变[①]，因此得到的结果会被打乱。前面展示的解决方案可避免出现这个问题。

本节中对生成器的使用反映出一个事实，那就是我们可能会希望这个函数尽可能的通用——不必绑定在只能对列表进行处理。比如，如果想读一个文件，去除其中重复的文本行，可以只需这样处理：

```
with open(somefile,'r') as f:
    for line in dedupe(f):
        ...
```

我们的 dedupe()函数也模仿了内置函数 sorted()、min()以及 max()对 key 函数的使用方式。例子可参考 1.8 节和 1.13 节。

1.11　对切片命名

1.11.1　问题

我们的代码已经变得无法阅读，到处都是硬编码的切片索引，我们想将它们清理干净。

1.11.2　解决方案

假设有一些代码用来从字符串的固定位置中取出具体的数据（比如从一个平面文件或

[①] 集合的特点就是集合中的元素都是唯一的，但不保证它们之间的顺序。——译者注

类似的格式）[1]：

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '....................100 .......513.25 ..........'
cost = int(record[20:32]) * float(record[40:48])
```

与其这样做，为什么不对切片命名呢？

```
SHARES = slice(20,32)
PRICE = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

在后一种版本中，由于避免了使用许多神秘难懂的硬编码索引，我们的代码就变得清晰了许多。

1.11.3 讨论

作为一条基本准则，代码中如果有很多硬编码的索引值，将导致可读性和可维护性都不佳。例如，如果一年以后再回过头来看代码，你会发现自己很想知道当初编写这些代码时自己在想些什么。前面展示的方法可以让我们对代码的功能有着更加清晰的认识。

一般来说，内置的 slice()函数会创建一个切片对象，可以用在任何允许进行切片操作的地方。例如：

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

如果有一个 slice 对象的实例 s，可以分别通过 s.start、s.stop 以及 s.step 属性来得到关于该对象的信息。例如：

```
>>> a = slice(
>>> a.start
10
>>> a.stop
```

[1] 平面文件（flat file）是一种包含没有相对关系结构的记录文件。——译者注

```
50
>>> a.step
2
>>>
```

此外，可以通过使用 indices(size)方法将切片映射到特定大小的序列上。这会返回一个(start, stop, step)元组，所有的值都已经恰当地限制在边界以内（当做索引操作时可避免出现 IndexError 异常）。例如：

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
... print(s[i])
...
W
r
d
>>>
```

1.12 找出序列中出现次数最多的元素

1.12.1 问题

我们有一个元素序列，想知道在序列中出现次数最多的元素是什么。

1.12.2 解决方案

collections 模块中的 Counter 类正是为此类问题所设计的。它甚至有一个非常方便的 most_common()方法可以直接告诉我们答案。

为了说明用法，假设有一个列表，列表中是一系列的单词，我们想找出哪些单词出现的最为频繁。下面是我们的做法：

```
words = [
   'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
   'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
   'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
   'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

1.12.3 讨论

可以给 Counter 对象提供任何可哈希的对象序列作为输入。在底层实现中，Counter 是一个字典，在元素和它们出现的次数间做了映射。例如：

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

如果想手动增加计数，只需简单地自增即可：

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
... word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

另一种方式是使用 update()方法。

```
>>> word_counts.update(morewords)
>>>
```

关于 Counter 对象有一个不为人知的特性，那就是它们可以轻松地同各种数学运算操作结合起来使用。例如：

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
         'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
         'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
         'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
```

```
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1})
>>>
```

不用说，当面对任何需要对数据制表或计数的问题时，Counter 对象都是你手边的得力工具。比起利用字典自己手写算法，更应该采用这种方式完成任务。

1.13 通过公共键对字典列表排序

1.13.1 问题

我们有一个字典列表，想根据一个或多个字典中的值来对列表排序。

1.13.2 解决方案

利用 operator 模块中的 itemgetter 函数对这类结构进行排序是非常简单的。假设通过查询数据库表项获取网站上的成员列表，我们得到了如下的数据结构：

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

根据所有的字典中共有的字段来对这些记录排序是非常简单的，示例如下：

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

以上代码的输出为：

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

itemgetter()函数还可以接受多个键。例如下面这段代码：

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

这会产生如下的输出：

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

1.13.3 讨论

在这个例子中，rows 被传递给内建的 sorted()函数，该函数接受一个关键字参数 key。这个参数应该代表一个可调用对象（callable），该对象从 rows 中接受一个单独的元素作为输入并返回一个用来做排序依据的值。itemgetter()函数创建的就是这样一个可调用对象。

函数 operator.itemgetter()接受的参数可作为查询的标记，用来从 rows 的记录中提取出所需要的值。它可以是字典的键名称、用数字表示的列表元素或是任何可以传给对象的__getitem__()方法的值。如果传多个标记给 itemgetter()，那么它产生的可调用对象将返回一个包含所有元素在内的元组，然后 sorted()将根据对元组的排序结果来排列输出结果。如果想同时针对多个字段做排序（比如例子中的姓和名），那么这是非常有用的。

有时候会用 lambda 表达式来取代 itemgetter()的功能。例如：

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

这种解决方案通常也能正常工作。但是用 itemgetter()通常会运行得更快一些。因此如果需要考虑性能问题的话，应该使用 itemgetter()。

最后不要忘了本节中所展示的技术同样适用于 min()和 max()这样的函数。例如：

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14 对不原生支持比较操作的对象排序

1.14.1 问题

我们想在同一个类的实例之间做排序，但是它们并不原生支持比较操作。

1.14.2 解决方案

内建的 sorted()函数可接受一个用来传递可调用对象（callable）的参数 key，而该可调用对象会返回待排序对象中的某些值，sorted 则利用这些值来比较对象。例如，如果应用中有一系列的 User 对象实例，而我们想通过 user_id 属性来对它们排序，则可以提供一个可调用对象将 User 实例作为输入然后返回 user_id。示例如下：

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>
```

除了可以用 lambda 表达式外，另一种方式是使用 operator.attrgetter()。

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

1.14.3 讨论

要使用 lambda 表达式还是 attrgetter()或许只是一种个人喜好。但是通常来说，attrgetter()要更快一些，而且具有允许同时提取多个字段值的能力。这和针对字典的 operator.itemgetter()的使用很类似（参见 1.13 节）。例如，如果 User 实例还有一个 first_name 和 last_name 属性的话，可以执行如下的排序操作：

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

同样值得一提的是，本节所用到的技术也适用于像 min()和 max()这样的函数。例如：

```
>>> min(users, key=attrgetter('user_id')
User(3)
>>> max(users, key=attrgetter('user_id')
User(99)
>>>
```

1.15 根据字段将记录分组

1.15.1 问题

有一系列的字典或对象实例，我们想根据某个特定的字段（比如说日期）来分组迭代数据。

1.15.2 解决方案

itertools.groupby()函数在对数据进行分组时特别有用。为了说明其用途，假设有如下的字典列表：

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

现在假设想根据日期以分组的方式迭代数据。要做到这些，首先以目标字段（在这个例子中是 date）来对序列排序，然后再使用 itertools.groupby()。

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

这会产生如下的输出：

```
07/01/2012
     {'date': '07/01/2012', 'address': '5412 N CLARK'}
     {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
     {'date': '07/02/2012', 'address': '5800 E 58TH'}
```

```
        {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
        {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
        {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
        {'date': '07/04/2012', 'address': '5148 N CLARK'}
        {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

1.15.3 讨论

函数 groupby()通过扫描序列找出拥有相同值（或是由参数 key 指定的函数所返回的值）的序列项，并将它们分组。groupby()创建了一个迭代器，而在每次迭代时都会返回一个值（value）和一个子迭代器（sub_iterator），这个子迭代器可以产生所有在该分组内具有该值的项。

在这里重要的是首先要根据感兴趣的字段对数据进行排序。因为 groupby()只能检查连续的项，不首先排序的话，将无法按所想的方式来对记录分组。

如果只是简单地根据日期将数据分组到一起，放进一个大的数据结构中以允许进行随机访问，那么利用 defaultdict()构建一个一键多值字典（multidict，见 1.6 节）可能会更好。例如：

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

这使得我们可以方便地访问每个日期的记录，如下所示：

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

对于后面这个例子，我们并不需要先对记录做排序。因此，如果不考虑内存方面的因素，这种方式会比先排序再用 groupby()迭代要来的更快。

1.16 筛选序列中的元素

1.16.1 问题

序列中含有一些数据，我们需要提取出其中的值或根据某些标准对序列做删减。

1.16.2 解决方案

要筛选序列中的数据，通常最简单的方法是使用列表推导式（list comprehension）。例如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

使用列表推导式的一个潜在缺点是如果原始输入非常大的话，这么做可能会产生一个庞大的结果。如果这是你需要考虑的问题，那么可以使用生成器表达式通过迭代的方式产生筛选的结果。例如：

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
1
4
10
2
3
>>>
```

有时候筛选的标准没法简单地表示在列表推导式或生成器表达式中。比如，假设筛选过程涉及异常处理或者其他一些复杂的细节。基于此，可以将处理筛选逻辑的代码放到单独的函数中，然后使用内建的 filter()函数处理。示例如下：

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

filter()创建了一个迭代器，因此如果我们想要的是列表形式的结果，请确保加上了 list()，就像示例中那样。

1.16.3 讨论

列表推导式和生成器表达式通常是用来筛选数据的最简单和最直接的方式。此外，它们也具有同时对数据做转换的能力。例如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

关于筛选数据，有一种情况是用新值替换掉不满足标准的值，而不是丢弃它们。例如，除了要找到正整数之外，我们也许还希望在指定的范围内将不满足要求的值替换掉。通常，这可以通过将筛选条件移到一个条件表达式中来轻松实现。就像下面这样：

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

另一个值得一提的筛选工具是 itertools.compress()，它接受一个可迭代对象以及一个布尔选择器序列作为输入。输出时，它会给出所有在相应的布尔选择器中为 True 的可迭代对象元素。如果想把对一个序列的筛选结果施加到另一个相关的序列上时，这就会非常有用。例如，假设有以下两列数据：

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK'
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

现在我们想构建一个地址列表，其中相应的 count 值要大于 5。下面是我们可以尝试的

方法：

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

这里的关键在于首先创建一个布尔序列，用来表示哪个元素可满足我们的条件。然后compress()函数挑选出满足布尔值为 True 的相应元素。

同 filter()函数一样，正常情况下 compress()会返回一个迭代器。因此，如果需要的话，得使用 list()将结果转为列表。

1.17 从字典中提取子集

1.17.1 问题

我们想创建一个字典，其本身是另一个字典的子集。

1.17.2 解决方案

利用字典推导式（dictionary comprehension）可轻松解决。例如：

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
   'IBM': 205.55,
   'HPQ': 37.20,
   'FB': 10.75
}

# Make a dictionary of all prices over 200
p1 = { key:value for key, value in prices.items() if value > 200 }

# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

1.17.3 讨论

大部分可以用字典推导式解决的问题也可以通过创建元组序列然后将它们传给 dict()函

数来完成。例如：

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

但是字典推导式的方案更加清晰，而且实际运行起来也要快很多（以本例中的字典 prices 来测试，效率要高 2 倍多）。

有时候会有多种方法来完成同一件事情。例如，第二个例子还可以重写成：

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

但是，计时测试表明这种解决方案几乎要比第一种慢上 1.6 倍。如果需要考虑性能因素，那么通常都需要花一点时间来研究它。有关计时和性能分析方面的信息，请参见 14.13 节。

1.18　将名称映射到序列的元素中

1.18.1　问题

我们的代码是通过位置（即索引，或下标）来访问列表或元组的，但有时候这会使代码变得有些难以阅读。我们希望可以通过名称来访问元素，以此减少结构中对位置的依赖性。

1.18.2　解决方案

相比普通的元组，collections.namedtuple()（命名元组）只增加了极小的开销就提供了这些便利。实际上 collections.namedtuple()是一个工厂方法，它返回的是 Python 中标准元组类型的子类。我们提供给它一个类型名称以及相应的字段，它就返回一个可实例化的类、为你已经定义好的字段传入值等。例如：

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

尽管 namedtuple 的实例看起来就像一个普通的类实例，但它的实例与普通的元组是可互换

的，而且支持所有普通元组所支持的操作，例如索引（indexing）和分解（unpacking）。比如：

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

命名元组的主要作用在于将代码同它所控制的元素位置间解耦。所以，如果从数据库调用中得到一个大型的元组列表，而且通过元素的位置来访问数据，那么假如在表单中新增了一列数据，那么代码就会崩溃。但如果首先将返回的元组转型为命名元组，就不会出现问题。

为了说明这个问题，下面有一些使用普通元组的代码：

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

通过位置来引用元素常常使得代码的表达力不够强，而且也很依赖于记录的具体结构。下面是使用命名元组的版本：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

当然，如果示例中的 records 序列已经包含了这样的实例，那么可以避免显式地将记录转换为 Stock 命名元组[①]。

1.18.3　讨论

namedtuple 的一种可能用法是作为字典的替代，后者需要更多的空间来存储。因此，如

[①] 作者的意思是如果 records 中的元素是某个类的实例，且已经有了 shares 和 price 这样的属性，那就可以直接通过属性名来访问，不需要通过位置来引用，也就没有必要再转换成命名元组了。——译者注

果要构建涉及字典的大型数据结构，使用 namedtuple 会更加高效。但是请注意，与字典不同的是，namedtuple 是不可变的（immutable）。例如：

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

如果需要修改任何属性，可以通过使用 namedtuple 实例的_replace()方法来实现。该方法会创建一个全新的命名元组，并对相应的值做替换。示例如下：

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

_replace()方法有一个微妙的用途，那就是它可以作为一种简便的方法填充具有可选或缺失字段的命名元组。要做到这点，首先创建一个包含默认值的“原型”元组，然后使用_replace()方法创建一个新的实例，把相应的值替换掉。示例如下：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
return stock_prototype._replace(**s)
```

让我们演示一下上面的代码是如何工作的：

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

最后，也是相当重要的是，应该要注意如果我们的目标是定义一个高效的数据结构，

而且将来会修改各种实例属性，那么使用 namedtuple 并不是最佳选择。相反，可以考虑定义一个使用__slots__属性的类（参见 8.4 节）。

1.19 同时对数据做转换和换算

1.19.1 问题

我们需要调用一个换算（reduction）函数（例如 sum()、min()、max()），但首先得对数据做转换或筛选。

1.19.2 解决方案

有一种非常优雅的方式能将数据换算和转换结合在一起——在函数参数中使用生成器表达式。例如，如果想计算平方和，可以像下面这样做：

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

这里还有一些其他的例子：

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')

# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))

# Data reduction across fields of a data structure
portfolio = [
   {'name':'GOOG', 'shares': 50},
   {'name':'YHOO', 'shares': 75},
   {'name':'AOL', 'shares': 20},
   {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

1.19.3 讨论

这种解决方案展示了当把生成器表达式作为函数的单独参数时在语法上的一些微妙之

处（即，不必重复使用括号）。比如，下面这两行代码表示的是同一个意思：

```
s = sum((x * x for x in nums))   # Pass generator-expr as argument
s = sum(x * x for x in nums)     # More elegant syntax
```

比起首先创建一个临时的列表，使用生成器做参数通常是更为高效和优雅的方式。例如，如果不使用生成器表达式，可能会考虑下面这种实现：

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

这也能工作，但这引入了一个额外的步骤而且创建了额外的列表。对于这么小的一个列表，这根本就无关紧要，但是如果 nums 非常巨大，那么就会创建一个庞大的临时数据结构，而且只用一次就要丢弃。基于生成器的解决方案可以以迭代的方式转换数据，因此在内存使用上要高效得多。

某些特定的换算函数比如 min()和 max()都可接受一个 key 参数，当可能倾向于使用生成器时会很有帮助。例如在 portfolio 的例子中，也许会考虑下面这种替代方案：

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)

# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20 将多个映射合并为单个映射

1.20.1 问题

我们有多个字典或映射，想在逻辑上将它们合并为一个单独的映射结构，以此执行某些特定的操作，比如查找值或检查键是否存在。

1.20.2 解决方案

假设有两个字典：

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

现在假设想执行查找操作，我们必须得检查这两个字典（例如，先在 a 中查找，如果没找到再去 b 中查找）。一种简单的方法是利用 collections 模块中的 ChainMap 类来解决这个问题。例如：

```
from collections import ChainMap
c = ChainMap(a,b)
```

```
print(c['x']) # Outputs 1 (from a)
print(c['y']) # Outputs 2 (from b)
print(c['z']) # Outputs 3 (from a)
```

1.20.3 讨论

ChainMap 可接受多个映射然后在逻辑上使它们表现为一个单独的映射结构。但是，这些映射在字面上并不会合并在一起。相反，ChainMap 只是简单地维护一个记录底层映射关系的列表，然后重定义常见的字典操作来扫描这个列表。大部分的操作都能正常工作。例如：

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

如果有重复的键，那么这里会采用第一个映射中所对应的值。因此，例子中的 c['z'] 总是引用字典 a 中的值，而不是字典 b 中的值。

修改映射的操作总是会作用在列出的第一个映射结构上。例如：

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap 与带有作用域的值，比如编程语言中的变量（即全局变量、局部变量等）一起工作时特别有用。实际上这里有一些方法使这个过程变得简单：

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
```

```
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

作为 ChainMap 的替代方案，我们可能会考虑利用字典的 update()方法将多个字典合并在一起。例如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

这么做行得通，但这需要单独构建一个完整的字典对象（或者修改其中现有的一个字典，这就破坏了原始数据）。此外，如果其中任何一个原始字典做了修改，这个改变都不会反应到合并后的字典中。例如：

```
>>> a['x'] = 13
>>> merged['x']
1
```

而 ChainMap 使用的就是原始的字典，因此它不会产生这种令人不悦的行为。示例如下：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']    # Notice change to merged dicts
42
>>>
```

第 2 章

字符串和文本

无论是解析数据还是产生输出，几乎每一个有实用价值的程序都会涉及某种形式的文本处理。本章的重点放在有关文本操作的常见问题上，例如拆分字符串、搜索、替换、词法分析以及解析。许多任务都可以通过内建的字符串方法轻松解决。但是，更复杂的操作可能会需要用到正则表达式或者创建完整的解析器才能得到解决。以上所有主题本章都有涵盖。此外，本章还提到了一些同 Unicode 打交道时用到的技巧。

2.1 针对任意多的分隔符拆分字符串

2.1.1 问题

我们需要将字符串拆分为不同的字段，但是分隔符（以及分隔符之间的空格）在整个字符串中并不一致。

2.1.2 解决方案

字符串对象的 split()方法只能处理非常简单的情况，而且不支持多个分隔符，对分隔符周围可能存在的空格也无能为力。当需要一些更为灵活的功能时，应该使用 re.split()方法：

```
>>> line = 'asdf fjdk; afed, fjek,asdf,     foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

2.1.3 讨论

re.split()是很有用的，因为可以为分隔符指定多个模式。例如，在上面的解决方案中，

分隔符可以是逗号、分号或者是空格符（后面可跟着任意数量的额外空格）。只要找到了对应的模式，无论匹配点的两端是什么字段，整个匹配的结果就成为那个分隔符。最终得到的结果是字段列表，同 str.split()得到的结果一样。

当使用 re.split()时，需要小心正则表达式模式中的捕获组（capture group）是否包含在了括号中。如果用到了捕获组，那么匹配的文本也会包含在最终结果中。比如，看看下面的结果：

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```

在特定的上下文中获取到分隔字符也可能是有用的。例如，也许稍后要用到分隔字符来改进字符串的输出：

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']

>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

如果不想在结果中看到分隔字符，但仍然想用括号来对正则表达式模式进行分组，请确保用的是非捕获组，以(?:...)的形式指定。示例如下：

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf','fjdk','afed','fjek','asdf','foo']
>>>
```

2.2 在字符串的开头或结尾处做文本匹配

2.2.1 问题

我们需要在字符串的开头或结尾处按照指定的文本模式做检查，例如检查文件的扩展名、URL 协议类型等。

2.2.2 解决方案

有一种简单的方法可用来检查字符串的开头或结尾，只要使用 str.startswith()和

str.endswith()方法就可以了。示例如下：

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

如果需要同时针对多个选项做检查，只需给 startswith()和 endswith()提供包含可能选项的元组即可：

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

这里有另一个例子：

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
             return f.read()
```

奇怪的是，这是 Python 中需要把元组当成输入的一个地方。如果我们刚好把选项指定在了列表或集合中，请确保首先用 tuple()将它们转换成元组。示例如下：

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

2.2.3 讨论

startswith()和 endswith()方法提供了一种非常方便的方式来对字符串的前缀和后缀做基本的检查。类似的操作也可以用切片来完成，但是那种方案不够优雅。例如：

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

可能我们也比较倾向于使用正则表达式作为替代方案。例如：

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

这也行得通，但是通常对于简单的匹配来说有些过于重量级了。使用本节提到的技术会更简单，运行得也更快。

最后但同样重要的是，当 startswith()和 endswith()方法和其他操作（比如常见的数据整理操作）结合起来时效果也很好。例如，下面的语句检查目录中有无出现特定的文件：

```
if any(name.endswith(('.c', '.h')) for name in listdir(dirname)):
...
```

2.3 利用 Shell 通配符做字符串匹配

2.3.1 问题

当工作在 UNIX Shell 下时，我们想使用常见的通配符模式（即，*.py、Dat[0-9]*.csv 等）来对文本做匹配。

2.3.2 解决方案

fnmatch 模块提供了两个函数——fnmatch()和 fnmatchcase()——可用来执行这样的匹配。使用起来很简单：

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
```

```
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
 True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

一般来说，fnmatch()的匹配模式所采用的大小写区分规则和底层文件系统相同（根据操作系统的不同而有所不同）。例如：

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False

>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

如果这个区别对我们而言很重要，就应该使用 fnmatchcase()。它完全根据我们提供的大小写方式来匹配：

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

关于这些函数，一个常被忽略的特性是它们在处理非文件名式的字符串时的潜在用途。例如，假设有一组街道地址，就像这样：

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

可以像下面这样写列表推导式：

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

2.3.3 讨论

fnmatch 所完成的匹配操作有点介乎于简单的字符串方法和全功能的正则表达式之间。如果只是试着在处理数据时提供一种简单的机制以允许使用通配符，那么通常这都是个合理的解决方案。

如果实际上是想编写匹配文件名的代码，那应该使用 glob 模块来完成，请参见 5.13 节。

2.4 文本模式的匹配和查找

2.4.1 问题

我们想要按照特定的文本模式进行匹配或查找。

2.4.2 解决方案

如果想要匹配的只是简单的文字，那么通常只需要用基本的字符串方法就可以了，比如 str.find()、str.endswith()、str.startswith()或类似的函数。示例如下：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Exact match
>>> text == 'yeah'
False

>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False

>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

对于更为复杂的匹配则需要使用正则表达式以及 re 模块。为了说明使用正则表达式的基本流程，假设我们想匹配以数字形式构成的日期，比如“11/27/2012”。示例如下：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
```

```
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
...     print('yes')
... else:
... print('no')
...
no
>>>
```

如果打算针对同一种模式做多次匹配，那么通常会先将正则表达式模式预编译成一个模式对象。例如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if datepat.match(text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

match()方法总是尝试在字符串的开头找到匹配项。如果想针对整个文本搜索出所有的匹配项，那么就应该使用 findall()方法。例如：

```
 >>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

当定义正则表达式时，我们常会将部分模式用括号包起来的方式引入捕获组。例如：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

捕获组通常能简化后续对匹配文本的处理，因为每个组的内容都可以单独提取出来。

例如：

```
>>> m = datepat.match('11/27/2012')
>>> m

<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>

>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

findall()方法搜索整个文本并找出所有的匹配项然后将它们以列表的形式返回。如果想以迭代的方式找出匹配项，可以使用 finditer()方法。示例如下：

```
>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>
```

2.4.3 讨论

有关正则表达式的基本理论教学超出了本书的范围。但是，本节向您展示了利用 re 模

块来对文本做匹配和搜索的基础。基本功能是首先用 re.compile()对模式进行编译，然后使用像 match()、findall()或 finditer()这样的方法做匹配和搜索。

当指定模式时我们通常会使用原始字符串，比如 r'(\d+)/(\d+)/(\d+)'。这样的字符串不会对反斜线字符转义，这在正则表达式上下文中会很有用。否则，我们需要用双反斜线来表示一个单独的'\'，例如'(\\d+)/(\\d+)/(\\d+)'。

请注意 match()方法只会检查字符串的开头。有可能出现匹配的结果并不是你想要的情况。例如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

如果想要精确匹配，请确保在模式中包含一个结束标记（$），示例如下：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最后，如果只是想执行简单的文本匹配和搜索操作，通常可以省略编译步骤，直接使用 re 模块中的函数即可。例如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

请注意，如果打算执行很多匹配或查找操作的话，通常需要先将模式编译然后再重复使用。模块级的函数会对最近编译过的模式做缓存处理，因此这里并不会有巨大的性能差异。但是使用自己编译过的模式会省下一些查找步骤和额外的处理。

2.5 查找和替换文本

2.5.1 问题

我们想对字符串中的文本做查找和替换。

2.5.2 解决方案

对于简单的文本模式，使用 str.replace()即可。例如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

针对更为复杂的模式，可以使用 re 模块中的 sub()函数/方法。为了说明如何使用，假设我们想把日期格式从“11/27/2012”改写为“2012-11-27”。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

sub()的第 1 个参数是要匹配的模式，第 2 个参数是要替换上的模式。类似“\3”这样的反斜线加数字的符号代表着模式中捕获组的数量。

如果打算用相同的模式执行重复替换，可以考虑先将模式编译以获得更好的性能。示例如下：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

对于更加复杂的情况，可以指定一个替换回调函数。示例如下：

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

替换回调函数的输入参数是一个匹配对象，由 match()或 find()返回。用.group()方法来提取匹配中特定的部分。这个函数应该返回替换后的文本。

除了得到替换后的文本外，如果还想知道一共完成了多少次替换，可以使用 re.subn()。例如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
```

```
>>> n
2
>>>
```

2.5.3 讨论

除了以上展示的 sub()调用之外，关于正则表达式的查找和替换并没有什么更多可说的了。最有技巧性的地方在于指定正则表达式模式——这个最好还是留给读者自己去练习吧。

2.6 以不区分大小写的方式对文本做查找和替换

2.6.1 问题

我们需要以不区分大小写的方式在文本中进行查找，可能还需要做替换。

2.6.2 解决方案

要进行不区分大小写的文本操作，我们需要使用 re 模块并且对各种操作都要加上 re.IGNORECASE 标记。例如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

上面这个例子揭示出了一种局限，那就是待替换的文本与匹配的文本大小写并不吻合。如果想修正这个问题，需要用到一个支撑函数（support function），示例如下：

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
return replace
```

下面是使用这个函数的例子：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

2.6.3 讨论

对于简单的情况,只需加上 re.IGNORECASE 标记就足以进行不区分大小写的匹配操作了。但请注意的是这对于某些涉及大写转换（case folding）的 Unicode 匹配来说可能是不够的。具体细节请参见 2.10 节。

2.7 定义实现最短匹配的正则表达式

2.7.1 问题

我们正在尝试用正则表达式对文本模式做匹配，但识别出来的是最长的可能匹配。相反，我们想将其修改为找出最短的可能匹配。

2.7.2 解决方案

这个问题通常会在匹配的文本被一对开始和结束分隔符包起来的时候出现（例如带引号的字符串）。为了说明这个问题，请看下面的例子：

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

在这个例子中，模式 r'\"(.*)\"'尝试去匹配包含在引号中的文本。但是，*操作符在正则表达式中采用的是贪心策略，所以匹配过程是基于找出最长的可能匹配来进行的。因此，在 text2 的例子中，它错误地匹配成 2 个被引号包围的字符串。

要解决这个问题，只要在模式中的*操作符后加上？修饰符就可以了。示例如下：

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

这么做使得匹配过程不会以贪心方式进行，也就会产生出最短的匹配了。

2.7.3 讨论

本节提到了一个当编写含有句点（.）字符的正则表达式时常会遇到的问题。在模式中，句点除了换行符之外可匹配任意字符。但是，如果以开始和结束文本（比如说引号）将句点括起来的话，在匹配过程中将尝试找出最长的可能匹配结果。这会导致匹配时跳过多个开始或结束文本，而将它们都包含在最长的匹配中。在*或+后添加一个？，会强制将匹配算法调整为寻找最短的可能匹配。

2.8 编写多行模式的正则表达式

2.8.1 问题

我们打算用正则表达式对一段文本块做匹配，但是希望在进行匹配时能够跨越多行。

2.8.2 解决方案

这个问题一般出现在希望使用句点（.）来匹配任意字符，但是忘记了句点并不能匹配换行符时。例如，假设想匹配 C 语言风格的注释：

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
...               multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

要解决这个问题，可以添加对换行符的支持。示例如下：

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n      multiline comment ']
>>>
```

在这个模式中，(?:.|\n)指定了一个非捕获组（即，这个组只做匹配但不捕获结果，也不会分配组号）。

2.8.3 讨论

re.compile()函数可接受一个有用的标记——re.DOTALL。这使得正则表达式中的句点(.)

可以匹配所有的字符，也包括换行符。例如：

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n        multiline comment ']
```

对于简单的情况，使用 re.DOTALL 标记就可以很好地完成工作。但是如果要处理极其复杂的模式，或者面对的是如 2.18 节中所描述的为了做分词（tokenizing）而将单独的正则表达式合并在一起的情况，如果可以选择的话，通常更好的方法是定义自己的正则表达式模式，这样它无需额外的标记也能正确工作。

2.9 将 Unicode 文本统一表示为规范形式

2.9.1 问题

我们正在同 Unicode 字符串打交道，但需要确保所有的字符串都拥有相同的底层表示。

2.9.2 解决方案

在 Unicode 中，有些特定的字符可以被表示成多种合法的代码点序列。为了说明这个问题，请看下面的示例：

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

这里的文本 “Spicy Jalapeño” 以两种形式呈现。第一种使用的是字符“ñ”的全组成（fully composed）形式（U+00F1）。第二种使用的是拉丁字母“n”紧跟着一个“~”组合而成的字符（U+0303）。

对于一个比较字符串的程序来说，同一个文本拥有多种不同的表示形式是个大问题。为了解决这个问题，应该先将文本统一表示为规范形式，这可以通过 unicodedata 模块

来完成：

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

normalize()的第一个参数指定了字符串应该如何完成规范表示。NFC 表示字符应该是全组成的（即，如果可能的话就使用单个代码点）。NFD 表示应该使用组合字符，每个字符应该是能完全分解开的。

Python 还支持 NFKC 和 NFKD 的规范表示形式，它们为处理特定类型的字符增加了额外的兼容功能。例如：

```
>>> s = '\ufb01' # A single character
>>> s
'ﬁ'
>>> unicodedata.normalize('NFD', s)
'ﬁ'

# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

2.9.3 讨论

对于任何需要确保以规范和一致性的方式处理 Unicode 文本的程序来说，规范化都是重要的一部分。尤其是在处理用户输入时接收到的字符串时，此时你无法控制字符串的编码形式，那么规范化文本的表示就显得更为重要了。

在对文本进行过滤和净化时，规范化同样也占据了重要的部分。例如，假设想从某些

文本中去除所有的音符标记（可能是为了进行搜索或匹配）：

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

最后一个例子展示了 unicodedata 模块的另一个重要功能——用来检测字符是否属于某个字符类别。使用工具 combining()函数可对字符做检查，判断它是否为一个组合型字符。这个模块中还有一些函数可用来查找字符类别、检测数字字符等。

很显然，Unicode 是一个庞大的主题。要获得更多有关规范化文本方面的参考信息，可访问 http://www.unicode.org/faq/normalization.html。Ned Batchelder 也在他的网站 http://nedbatchelder.com/text/unipain.html 上对 Python 中的 Unicode 处理给出了优秀的示例说明。

2.10 用正则表达式处理 Unicode 字符

2.10.1 问题

我们正在用正则表达式处理文本，但是需要考虑处理 Unicode 字符。

2.10.2 解决方案

默认情况下 re 模块已经对某些 Unicode 字符类型有了基本的认识。例如，\d 已经可以匹配任意 Unicode 数字字符了：

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>

>>> # Arabic digits
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

如果需要在模式字符串中包含指定的 Unicode 字符，可以针对 Unicode 字符使用转义序列（例如\uFFFF 或\UFFFFFFF）。比如，这里有一个正则表达式能在多个不同的阿拉伯代码页中匹配所有的字符：

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
>>>
```

当执行匹配和搜索操作时，一个好主意是首先将所有的文本都统一表示为标准形式（见 2.9 节）。但是，同样重要的是需要注意一些特殊情况。例如，当不区分大小写的匹配

和大写转换（case folding）匹配联合起来时，考虑会出现什么行为：

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'stra遻'
>>> pat.match(s)              # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper())      # Doesn't match
>>> s.upper()                 # Case folds
'STRASSE'
>>>
```

2.10.3 讨论

把 Unicode 和正则表达式混在一起使用绝对是个能让人头痛欲裂的办法。如果真的要这么做，应该考虑安装第三方的正则表达式库（http://pypi.python.org/pypi/regex），这些第三方库针对 Unicode 大写转换提供了完整的支持，还包含其他各种有趣的特性，包括近似匹配。

2.11 从字符串中去掉不需要的字符

2.11.1 问题

我们想在字符串的开始、结尾或中间去掉不需要的字符，比如说空格符。

2.11.2 解决方案

strip()方法可用来从字符串的开始和结尾处去掉字符。lstrip()和 rstrip()可分别从左或从右侧开始执行去除字符的操作。默认情况下这些方法去除的是空格符，但也可以指定其他的字符。例如：

```
>>> # Whitespace stripping
>>> s = ' hello world \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>

>>> # Character stripping
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

2.11.3 讨论

当我们读取并整理数据以待稍后的处理时常常会用到这类 strip()方法。例如，可以用它们来去掉空格、移除引号等。

需要注意的是，去除字符的操作并不会对位于字符串中间的任何文本起作用。例如：

```
>>> s = ' hello    world    \n'
>>> s = s.strip()
>>> s
'hello     world'
>>>
```

如果要对里面的空格执行某些操作，应该使用其他技巧，比如使用 replace()方法或正则表达式替换。例如：

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

我们通常会遇到的情况是将去除字符的操作同某些迭代操作结合起来，比如说从文件中读取文本行。如果是这样的话，那就到了生成器表达式大显身手的时候了。例如：

```
with open(filename) as f:
lines = (line.strip() for line in f)
for line in lines:
        ...
```

这里，表达式 lines = (line.strip() for line in f)的作用是完成数据的转换[①]。它很高效，因为这里并没有先将数据读取到任何形式的临时列表中。它只是创建一个迭代器，在所有产生出的文本行上都会执行 strip 操作。

对于更高级的 strip 操作，应该转而使用 translate()方法。请参见下一节以获得进一步的细节。

2.12 文本过滤和清理

2.12.1 问题

某些无聊的脚本小子在 Web 页面表单中填入了“pýtĥöñ”这样的文本，我们想以某种方式将其清理掉。

① 把原始数据中每一行开头和结尾处的空格符去掉，相当于一种转换处理。——译者注

2.12.2 解决方案

文本过滤和清理所涵盖的范围非常广泛，涉及文本解析和数据处理方面的问题。在非常简单的层次上，我们可能会用基本的字符串函数（例如 str.upper()和 str.lower()）将文本转换为标准形式。简单的替换操作可通过 str.replace()或 re.sub()来完成，它们把重点放在移除或修改特定的字符序列上。也可以利用 unicodedata.normalize()来规范化文本，如 2.9 节所示。

然而我们可能想更进一步。比方说也许想清除整个范围内的字符，或者去掉音符标志。要完成这些任务，可以使用常被忽视的 str.translate()方法。为了说明其用法，假设有如下这段混乱的字符串：

```
>>> s = 'pýtĥöñ\fis\tawesome\r\n'
>>> s
'pýtĥöñ\x0cis\tawesome\r\n'
>>>
```

第一步是清理空格。要做到这步，先建立一个小型的转换表，然后使用 translate()方法：

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None      # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýtĥöñ is awesome\n'
>>>
```

可以看到，类似\t 和\f 这样的空格符已经被重新映射成一个单独的空格。回车符\r 已经完全被删除掉了。

可以利用这种重新映射的思想进一步构建出更加庞大的转换表。例如，我们把所有的 Unicode 组合字符都去掉：

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýtĥöñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

在这个例子中，我们使用 dict.fromkeys()方法构建了一个将每个 Unicode 组合字符都映射为 None 的字典。

原始输入会通过 unicodedata.normalize()方法转换为分离形式，然后再通过 translate()方法删除所有的重音符号。我们也可以利用相似的技术来去掉其他类型的字符（例如控制字符）。

下面来看另一个例子。这里有一张转换表将所有的 Unicode 十进制数字字符映射为它们对应的 ASCII 版本：

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...              for c in range(sys.maxunicode)
...              if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>
```

另一种用来清理文本的技术涉及 I/O 解码和编码函数。大致思路是首先对文本做初步的清理，然后通过结合 encode()和 decode()操作来修改或清理文本。示例如下：

```
>>> a
'pýtĥöñ is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

这里的 normalize()方法先对原始文本做分解操作。后续的 ASCII 编码/解码只是简单地一次性丢弃所有不需要的字符。很显然，这种方法只有当我们的最终目标就是 ASCII 形式的文本时才有用。

2.12.3 讨论

文本过滤和清理的一个主要问题就是运行时的性能。一般来说操作越简单，运行得就越快。对于简单的替换操作，用 str.replace()通常是最快的方式——即使必须多次调用它也是如此。比方说如果要清理掉空格符，可以编写如下的代码：

```
def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
```

```
        s = s.replace('\f', ' ')
    return s
```

如果试着调用它，就会发现这比使用 translate()或者正则表达式的方法要快得多。

另一方面,如果需要做任何高级的操作,比如字符到字符的重映射或删除,那么 translate()方法还是非常快的。

从整体来看，我们应该在具体的应用中去进一步揣摩性能方面的问题。不幸的是，想在技术上给出一条“放之四海而皆准”的建议是不可能的，所以应该尝试多种不同的方法，然后做性能统计分析。

尽管本节的内容主要关注的是文本，但类似的技术也同样适用于字节对象（byte），这包括简单的替换、翻译和正则表达式。

2.13 对齐文本字符串

2.13.1 问题

我们需要以某种对齐方式将文本做格式化处理。

2.13.2 解决方案

对于基本的字符串对齐要求，可以使用字符串的 ljust()、rjust()和 center()方法。示例如下：

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World '
>>> text.rjust(20)
'         Hello World'
>>> text.center(20)
'    Hello World '
>>>
```

所有这些方法都可接受一个可选的填充字符。例如：

```
>>> text.rjust(20,'=')
'=========Hello World'
>>> text.center(20,'*')
'****Hello World*****'
>>>
```

format()函数也可以用来轻松完成对齐的任务。需要做的就是合理利用'<'、'>'，或'^'字

符以及一个期望的宽度值[①]。例如：

```
>>> format(text, '>20')
'         Hello World'
>>> format(text, '<20')
'Hello World         '
>>> format(text, '^20')
'    Hello World     '
>>>
```

如果想包含空格之外的填充字符，可以在对齐字符之前指定：

```
>>> format(text, '=>20s')
'=========Hello World'
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

当格式化多个值时，这些格式化代码也可以用在 format()方法中。例如：

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'     Hello      World'
>>>
```

format()的好处之一是它并不是特定于字符串的。它能作用于任何值，这使得它更加通用。例如，可以对数字做格式化处理：

```
>>> x = 1.2345
>>> format(x, '>10')
'    1.2345'
>>> format(x, '^10.2f')
'   1.23   '
>>>
```

2.13.3 讨论

在比较老的代码中，通常会发现%操作符用来格式化文本。例如：

```
>>> '%-20s' % text
'Hello World         '
>>> '%20s' % text
'         Hello World'
>>>
```

但是在新的代码中，我们应该会更钟情于使用 format()函数或方法。format()比%操作符提供的功能要强大多了。此外，format()可作用于任意类型的对象，比字符串的 ljust()、rjust()以及 center()方法要更加通用。

---

① '>'表示右对齐，'<'表示左对齐，'^'表示居中对齐，这些字符称为对齐字符。——译者注

想了解 format()函数的所有功能,请参考 Python 的在线手册 http://docs.python.org/3/ library/ string. html#formatspec。

2.14 字符串连接及合并

2.14.1 问题

我们想将许多小字符串合并成一个大的字符串。

2.14.2 解决方案

如果想要合并的字符串在一个序列或可迭代对象中，那么将它们合并起来的最快方法就是使用 join()方法。示例如下：

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

初看上去语法可能显得有些怪异，但是 join()操作其实是字符串对象的一个方法。这么设计的部分原因是因为想要合并在一起的对象可能来自于各种不同的数据序列，比如列表、元组、字典、文件、集合或生成器，如果单独在每一种序列对象中实现一个 join()方法就显得太冗余了。因此只需要指定想要的分隔字符串，然后在字符串对象上使用 join()方法将文本片段粘合在一起就可以了。

如果只是想连接一些字符串，一般使用+操作符就足够完成任务了：

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

针对更加复杂的字符串格式化操作，+操作符同样可以作为 format()的替代，很好地完成任务：

```
>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

如果打算在源代码中将字符串字面值合并在一起，可以简单地将它们排列在一起，中间不加+操作符。示例如下：

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

2.14.3 讨论

字符串连接这个主题可能看起来还没有高级到要用一整节的篇幅来讲解，但是程序员常常会在这个问题上做出错误的编程选择，使得他们的代码性能受到影响。

最重要的一点是要意识到使用+操作符做大量的字符串连接是非常低效的，原因是由于内存拷贝和垃圾收集产生的影响。特别是你绝不会想写出这样的字符串连接代码：

```
s = ''
for p in parts:
    s += p
```

这种做法比使用 join()方法要慢上许多。主要是因为每个+=操作都会创建一个新的字符串对象。我们最好先收集所有要连接的部分，最后再一次将它们连接起来。

一个相关的技巧（很漂亮的技巧）是利用生成器表达式（见 1.19 节）在将数据转换为字符串的同时完成连接操作。示例如下：

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

对于不必要的字符串连接操作也要引起重视。有时候在技术上并非必需的时候，程序员们也会忘乎所以地使用字符串连接操作。例如在打印的时候：

```
print(a + ':' + b + ':' + c)     # Ugly
print(':'.join([a, b, c]))      # Still ugly
print(a, b, c, sep=':')         # Better
```

将字符串连接同 I/O 操作混合起来的时候需要对应用做仔细的分析。例如，考虑如下两段代码：

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)
```

```
# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

如果这两个字符串都很小，那么第一个版本的代码能带来更好的性能，这是因为执行一次 I/O 系统调用的固有开销就很高。另一方面，如果这两个字符串都很大，那么第二个版本的代码会更加高效。因为这里避免了创建大的临时结果，也没有对大块的内存进行拷贝。这里必须再次强调，你需要对自己的数据做分析，以此才能判定哪一种方法可以获得最好的性能。

最后但也是最重要的是，如果我们编写的代码要从许多短字符串中构建输出，则应该考虑编写生成器函数，通过 yield 关键字生成字符串片段。示例如下：

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

关于这种方法有一个有趣的事实，那就是它不会假设产生的片段要如何组合在一起。比如说可以用 join()将它们简单的连接起来：

```
text = ''.join(sample())
```

或者，也可以将这些片段重定向到 I/O：

```
for part in sample():
    f.write(part)
```

又或者我们能以混合的方式将 I/O 操作智能化地结合在一起：

```
def combine(source, maxsize):
     parts = []
     size = 0
     for part in source:
         parts.append(part)
         size += len(part)
         if size > maxsize:
             yield ''.join(parts)
             parts = []
             size = 0
    yield ''.join(parts)

for part in combine(sample(), 32768):
    f.write(part)
```

关键在于这里的生成器函数并不需要知道精确的细节，它只是产生片段而已。

2.15 给字符串中的变量名做插值处理

2.15.1 问题

我们想创建一个字符串，其中嵌入的变量名称会以变量的字符串值形式替换掉。

2.15.2 解决方案

Python 并不直接支持在字符串中对变量做简单的值替换。但是，这个功能可以通过字符串的 format()方法近似模拟出来。示例如下：

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

另一种方式是，如果要被替换的值确实能在变量中找到，则可以将 format_map()和 vars()联合起来使用，示例如下：

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

有关 vars()的一个微妙的特性是它也能作用于类实例上。比如：

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido',37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

而 format()和 format_map()的一个缺点则是没法优雅地处理缺少某个值的情况。例如：

```
>>> s.format(name='Guido')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

避免出现这种情况的一种方法就是单独定义一个带有__missing__()方法的字典类，示例如下：

```
class safesub(dict):
    def __missing__(self, key):
        return '{' + key + '}'
```

现在用这个类来包装传给 format_map()的输入参数：

```
>>> del n    # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

如果发现自己在代码中常常需要执行这些步骤，则可以将替换变量的过程隐藏在一个小型的功能函数内，这里要采用一种称之为“frame hack”的技巧[①]。示例如下：

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

现在，我们就可以像这样编写代码了：

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

2.15.3 讨论

多年来，由于 Python 缺乏真正的变量插值功能，由此产生了各种解决方案。作为本节中已给出的解决方案的替代，有时候我们会看到类似下面代码中的字符串格式化操作：

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
```

[①] 即需要同函数的栈帧打交道。sys._getframe 这个特殊的函数可以让我们获得调用函数的栈信息。——译者注

```
'Guido has 37 messages.'
>>>
```

我们可能还会看到模板字符串（template string）的使用：

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

但是，format()和 format_map()方法比上面这些替代方案都要更加现代化，我们应该将其作为首选。使用 format()的一个好处是可以同时得到所有关于字符串格式化方面的功能（对齐、填充、数值格式化等），而这些功能在字符串模板对象上是不可能做到的。

在本节的部分内容中还提到了一些有趣的高级特性。字典类中鲜为人知的__missing__()方法可用来处理缺少值时的行为。在 safesub 类中，我们将该方法定义为将缺失的值以占位符的形式返回，因此这里不会抛出 KeyError 异常，缺少的那个值会出现在最后生成的字符串中（可能对调试有些帮助）。

sub()函数使用了 sys._getframe(1)来返回调用方的栈帧。通过访问属性 f_locals 来得到局部变量。无需赘言，在大部分的代码中都应该避免去和栈帧打交道，但是对于类似完成字符串替换功能的函数来说，这会是有用的。插一句题外话，值得指出的是 f_locals 是一个字典，它完成对调用函数中局部变量的拷贝。尽管可以修改 f_locals 的内容，可是修改后并不会产生任何持续性的效果。因此，尽管访问不同的栈帧可能看起来是很邪恶的，但是想意外地覆盖或修改调用方的本地环境也是不可能的。

2.16 以固定的列数重新格式化文本

2.16.1 问题

我们有一些很长的字符串，想将它们重新格式化，使得它们能按照用户指定的列数来显示。

2.16.2 解决方案

可以使用 textwrap 模块来重新格式化文本的输出。例如，假设有如下这段长字符串：

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
```

```
look into my eyes, you're under."
```

这里可以用 textwrap 模块以多种方式来重新格式化字符串：

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent=' '))
Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent=' '))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not
around the eyes, don't look around
the eyes, look into my eyes, you're
under.
```

2.16.3 讨论

textwrap 模块能够以简单直接的方式对文本格式做整理使其适合于打印——尤其是当希望输出结果能很好地显示在终端上时。关于终端的尺寸大小，可以通过 os.get_terminal_size()来获取。例如：

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

fill()方法还有一些额外的选项可以用来控制如何处理制表符、句号等。请参阅 textwrap.TextWrapper 类的文档（http://docs.python.org/3.3/library/ textwrap. html# text w r ap.TextWrapper）以获得进一步的细节。

2.17　在文本中处理 HTML 和 XML 实体

2.17.1　问题

我们想将&entity 或&#code 这样的 HTML 或 XML 实体替换为它们相对应的文本。或者，我们需要生成文本，但是要对特定的字符（比如<,>或&）做转义处理。

2.17.2　解决方案

如果要生成文本，使用 html.escape()函数来完成替换<or>这样的特殊字符相对来说是比较容易的。例如：

```
>>> s = 'Elements are written as "<tag>text</tag>".'
>>> import html
>>> print(s)
Elements are written as "<tag>text</tag>".
>>> print(html.escape(s))
Elements are written as &quot;&lt;tag&gt;text&lt;/tag&gt;&quot;.

>>> # Disable escaping of quotes
>>> print(html.escape(s, quote=False))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
>>>
```

如果要生成 ASCII 文本，并且想针对非 ASCII 字符将它们对应的字符编码实体嵌入到文本中，可以在各种同 I/O 相关的函数中使用 errors='xmlcharrefreplace'参数来实现。示例如下：

```
>>> s = 'Spicy Jalapeño'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

要替换文本中的实体，那就需要不同的方法。如果实际上是在处理 HTML 或 XML，首先应该尝试使用一个合适的 HTML 或 XML 解析器。一般来说，这些工具在解析的过程中会自动处理相关值的替换，而我们完全无需为此操心。

如果由于某种原因在得到的文本中带有一些实体，而我们想手工将它们替换掉，通常可以利用各种 HTML 或 XML 解析器自带的功能函数和方法来完成。示例如下：

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
```

```
'Spicy "Jalapeño".'
>>>

>>> t = 'The prompt is &gt;&gt;&gt;'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

2.17.3 讨论

在生成 HTML 或 XML 文档时，适当地对特殊字符做转义处理常常是个容易被忽视的细节。尤其是当自己用 print()或其他一些基本的字符串格式化函数来产生这类输出时更是如此。简单的解决方案是使用像 html.escape()这样的工具函数。

如果需要反过来处理文本（即，将 HTML 或 XML 实体转换成对应的字符），有许多像 xml.sax.saxutils.unescape()这样的工具函数能帮上忙。但是，我们需要仔细考察一个合适的解析器应该如何使用。例如，如果是处理 HTML 或 XML，像 html.parser 或 xml.etree.ElementTree 这样的解析模块应该已经解决了有关替换文本中实体的细节问题。

2.18 文本分词

2.18.1 问题

我们有一个字符串，想从左到右将它解析为标记流（stream of tokens）。

2.18.2 解决方案

假设有如下的字符串文本：

```
text = 'foo = 23 + 42 * 10'
```

要对字符串做分词处理，需要做的不仅仅只是匹配模式。我们还需要有某种方法来识别出模式的类型。例如，我们可能想将字符串转换为如下的序列对：

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', 10')]
```

要完成这样的分词处理，第一步是定义出所有可能的标记，包括空格。这可以通过正则表达式中的命名捕获组来实现，示例如下：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
```

```
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ    = r'(?P<EQ>=)'
WS    = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

在这些正则表达式模式中，形如?P<TOKENNAME>这样的约定是用来将名称分配给该模式的。这个我们稍后会用到。

接下来我们使用模式对象的 scanner()方法来完成分词操作。该方法会创建一个扫描对象，在给定的文本中重复调用 match()，一次匹配一个模式。下面这个交互式示例展示了扫描对象是如何工作的：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

要利用这项技术并将其转化为代码，我们可以做些清理工作然后轻松地将其包含在一个生成器函数中，示例如下：

```
from collections import namedtuple

Token = namedtuple('Token', ['type','value'])
```

```
def generate_tokens(pat, text):
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
     print(tok)

# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

如果想以某种方式对标记流做过滤处理，要么定义更多的生成器函数，要么就用生成器表达式。例如，下面的代码告诉我们如何过滤掉所有的空格标记。

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

2.18.3 讨论

对于更加高级的文本解析，第一步往往是分词处理。要使用上面展示的扫描技术，有几个重要的细节需要牢记于心。第一，对于每个可能出现在输入文本中的文本序列，都要确保有一个对应的正则表达式模式可以将其识别出来。如果发现有任何不能匹配的文本，扫描过程就会停止。这就是为什么有必要在上面的示例中指定空格标记（WS）。

这些标记在正则表达式（即 re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))）中的顺序同样也很重要。当进行匹配时，re 模块会按照指定的顺序来对模式做匹配。因此，如果碰巧某个模式是另一个较长模式的子串时，就必须确保较长的那个模式要先做匹配。示例如下：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

第 2 个模式是错误的（注释掉的那一行），因为这样会把文本'<='匹配为 LT（'<'）紧跟

着 EQ（'='），而没有匹配为单独的标记 LE（'<='），这与我们的本意不符。

最后也最重要的是，对于有可能形成子串的模式要多加小心。例如，假设有如下两种模式：

```
PRINT = r'(P<PRINT>print)'
NAME  = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :
# Token(type='PRINT', value='print')
# Token(type='NAME', value='er')
```

对于更加高级的分词处理，我们应该去看看像 PyParsing 或 PLY 这样的包。有关 PLY 的例子将在下一节中讲解。

2.19 编写一个简单的递归下降解析器

2.19.1 问题

我们需要根据一组语法规则来解析文本，以此执行相应的操作或构建一个抽象语法树来表示输入。语法规则很简单，因此我们倾向于自己编写解析器而不是使用某种解析器框架。

2.19.2 解决方案

在这个问题中，我们把重点放在根据特定的语法来解析文本上。要做到这些，应该以 BNF 或 EBNF 的形式定义出语法的正式规格。比如，对于简单的算术运算表达式，语法看起来是这样的：

```
expr ::= expr + term
     | expr - term
     | term
term ::= term * factor
     | term / factor
     | factor
factor ::= ( expr )
       | NUM
```

又或者以 EBNF 的形式定义为如下形式：

```
expr ::= term { (+|-) term }*
term ::= factor { (*|/) factor }*
factor ::= ( expr )
     | NUM
```

在 EBNF 中，部分包括在{ ... }*中的规则是可选的。*意味着零个或更多重复项（和在正则表达式中的意义相同）。

现在，如果我们对 BNF 还不熟悉的话，可以把它看做是规则替换或取代的一种规范形式，左侧的符号可以被右侧的符号所取代（反之亦然）。一般来说，在解析的过程中我们会尝试将输入的文本同语法做匹配，通过 BNF 来完成各种替换和扩展。为了说明，假设正在解析一个类似于 3 + 4 * 5 这样的表达式。这个表达式首先应该被分解为标记流，这可以使用 2.18 节中描述的技术来实现。得到的结果可能是下面这样的标记序列：

```
NUM + NUM * NUM
```

从这里开始，解析过程就涉及通过替换的方式将语法匹配到输入标记上：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

完成所有的替换需要花上一段时间，这是由输入的规模和尝试去匹配的语法规则所决定的。第一个输入标记是一个 NUM，因此替换操作首先会把重点放在匹配这一部分上。一旦匹配上了，重点就转移到下一个标记+上，如此往复。当发现无法匹配下一个标记时，右手侧的特定部分（{ (*/) factor }*）就会消失。在一个成功的解析过程中，整个右手侧部分会完全根据匹配到的输入标记流来相应地扩展。

有了前面这些基础，下面就向各位展示如何构建一个递归下降的表达式计算器：

```
import re
import collections

# Token specification
NUM    = r'(?P<NUM>\d+)'
PLUS   = r'(?P<PLUS>\+)'
```

```
MINUS  = r'(?P<MINUS>-)'
TIMES  = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer
Token = collections.namedtuple('Token', ['type','value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser
class ExpressionEvaluator:
'''
Implementation of a recursive descent parser. Each method
implements a single grammar rule. Use the ._accept() method
to test and accept the current lookahead token. Use the ._expect()
method to exactly match and discard the next token on on the input
(or raise a SyntaxError if it doesn't match).
'''

def parse(self,text):
    self.tokens = generate_tokens(text)
    self.tok = None             # Last symbol consumed
    self.nexttok = None         # Next symbol tokenized
    self._advance()             # Load first lookahead token
    return self.expr()

def _advance(self):
    'Advance one token ahead'
    self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

def _accept(self,toktype):
    'Test and consume the next token if it matches toktype'
    if self.nexttok and self.nexttok.type == toktype:
        self._advance()
        return True
```

```
    else:
        return False

def _expect(self,toktype):
    'Consume next token if it matches toktype or raise SyntaxError'
     if not self._accept(toktype):
         raise SyntaxError('Expected ' + toktype)

# Grammar rules follow

def expr(self):
    "expression ::= term { ('+'|'-') term }*"

    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }*"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
```

```
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')
```

下面是以交互式的方式使用 ExpressionEvaluator 类的示例：

```
>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5
>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exprparse.py", line 40, in parse
    return self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 93, in factor
    exprval = self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 97, in factor
    raise SyntaxError("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>
```

如果我们想做的不只是纯粹的计算，那就需要修改 ExpressionEvaluator 类来实现。比如，下面的实现构建了一棵简单的解析树：

```
class ExpressionTreeBuilder(ExpressionEvaluator):
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
```

```
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                exprval = ('-', exprval, right)
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }"

        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval = ('*', termval, right)
            elif op == 'DIVIDE':
                termval = ('/', termval, right)
        return termval

    def factor(self):
        'factor ::= NUM | ( expr )'

        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')
```

下面的示例展示了它是如何工作的：

```
>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>
```

2.19.3 讨论

文本解析是一个庞大的主题，一般会占用学生们编译原理课程的前三周时间。如果你

正在寻找有关语法、解析算法和其他相关信息的背景知识，那么应该去找一本编译器方面的图书来读。无需赘言，本书是不会重复那些内容的。

然而，要编写一个递归下降的解析器，总体思路还是比较简单的。我们要将每一条语法规则转变为一个函数或方法。因此，如果我们的语法看起来是这样的：

```
expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
       | NUM
```

就可以像下面这样将它们转换为对应的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
        ...
    def term(self):
        ...

    def factor(self):
        ...
```

每个方法的任务很简单——必须针对语法规则的每个部分从左到右扫描，在扫描过程中处理符号标记。从某种意义上说，这些方法的目的就是顺利地将规则消化掉，如果卡住了就产生一个语法错误。要做到这点，需要应用下面这些实现技术。

- 如果规则中的下一个符号标记是另一个语法规则的名称（例如，term 或者 factor），就需要调用同名的方法。这就是算法中的“下降”部分——控制其下降到另一个语法规则中。有时候规则中会涉及调用已经在执行的方法（例如，在规则 factor ::= '(' expr ')'中对 expr 的调用）。这就是算法中的“递归”部分。
- 如果规则中的下一个符号标记是一个特殊的符号（例如'('），需要检查下一个标记，看它们是否能完全匹配。如果不能匹配，这就是语法错误。本节给出的_expect()方法就是用来处理这些步骤的。
- 如果规则中的下一个符号标记存在多种可能的选择（例如+或-），则必须针对每种可能性对下一个标记做检查，只有在有匹配满足时才前进到下一步。这就是本节给出的_accept()方法的目的所在。这有点像_except()的弱化版，在_accept()中如果有匹配满足，就前进到下一步，但如果没有匹配，它只是简单的回退而不会引发一个错误（这样检查才可以继续进行下去）。

- 对于语法规则中出现的重复部分（例如 expr ::=term { ('+' | '-') term }* ）,这是通过 while 循环来实现的。一般在循环体中收集或处理所有的重复项，直到无法找到更多的重复项为止。
- 一旦整个语法规则都已经处理完，每个方法就返回一些结果给调用者。这就是在解析过程中将值进行传递的方法。比如，在计算器表达式中，表达式解析的部分结果会作为值来返回。最终它们会结合在一起，在最顶层的语法规则方法中得到执行。

尽管本节给出的例子很简单，但递归下降解析器可以用来实现相当复杂的解析器。例如，Python 代码本身也是通过一个递归下降解析器来解释的。如果对此很感兴趣，可以通过检查 Python 源代码中的 Grammar/Grammar 文件来一探究竟。即便如此，要自己手写一个解析器时仍然需要面对各种陷阱和局限。

局限之一就是对于任何涉及左递归形式的语法规则，都没法用递归下降解析器来解决。例如，假设需要解释如下的规则：

```
items ::= items ',' item
       | item
```

要完成这样的解析，我们可能会试着这样来定义 items()方法：

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
         itemsval.append(self.item())
    else:
         itemsval = [ self.item() ]
```

唯一的问题就是这么做行不通。实际上这会产生一个无穷递归的错误。

我们也可能会陷入到语法规则自身的麻烦中。例如，我们可能想知道表达式是否能以这种加简单的语法形式来描述：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
        | NUM
```

这个语法从技术上说是能实现的，但是它却并没有遵守标准算术中关于计算顺序的约定。比如说，表达式“3 + 4 * 5”会被计算为 35，而不是我们预期的 23。因此这里需要单独的“expr”和“term”规则来确保计算结果的正确性。

对于真正复杂的语法解析，最好还是使用像 PyParsing 或 PLY 这样的解析工具。如果使用 PLY 的话，解析计算器表达式的代码看起来是这样的：

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]

# Ignored characters

t_ignore = ' \t\n'

# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
    | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    '''
```

```
        expr : term
        '''
        p[0] = p[1]

    def p_term(p):
        '''
        term : term TIMES factor
             | term DIVIDE factor
        '''
        if p[2] == '*':
            p[0] = p[1] * p[3]
        elif p[2] == '/':
            p[0] = p[1] / p[3]

    def p_term_factor(p):
        '''
        term : factor
        '''
        p[0] = p[1]

    def p_factor(p):
        '''
        factor : NUM
        '''
        p[0] = p[1]

    def p_factor_group(p):
        '''
        factor : LPAREN expr RPAREN
        '''
        p[0] = p[2]

    def p_error(p):
        print('Syntax error')

    parser = yacc()
```

在这份代码中会发现所有的东西都是以一种更高层的方式来定义的。我们只需编写匹配标记符号的正则表达式，以及当匹配各种语法规则时所需要的高层处理函数就行了。而实际运行解析器、接收符号标记等都完全由库来实现。

下面是如何使用解析器对象的示例：

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
```

```
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果想在编程中增加一点激动兴奋的感觉，编写解析器和编译器会是非常有趣的课题。再次说明，一本编译器方面的教科书会涵盖许多理论之下的底层细节。但是，在网上同样也能找到许多优秀的在线资源。Python 自带的 ast 模块也同样值得去看看。

2.20 在字节串上执行文本操作

2.20.1 问题

我们想在字节串（Byte String）上执行常见的文本操作（例如，拆分、搜索和替换）。

2.20.2 解决方案

字节串已经支持大多数和文本字符串一样的内建操作。例如：

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

类似这样的操作在字节数组上也能完成。例如：

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

我们可以在字节串上执行正则表达式的模式匹配操作，但是模式本身需要以字节串的

形式来指定。示例如下：

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/re.py", line 191, in split
    return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object

>>> re.split(b'[:,]',data)  # Notice: pattern as bytes
[b'FOO', b'BAR', b'SPAM']
>>>
```

2.20.3 讨论

就绝大部分情况而言，几乎所有能在文本字符串上执行的操作同样也可以在字节串上进行。但是，还是有几个显著的区别值得大家注意。例如：

```
>>> a = 'Hello World'      # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World'     # Byte string
>>> b[0]
72
>>> b[1]
101
>>>
```

这种语义上的差异会对试图按照字符的方式处理面向字节流数据的程序带来影响。

其次，字节串并没有提供一个漂亮的字符串表示，因此打印结果并不干净利落，除非首先将其解码为文本字符串。示例如下：

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World'                # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

同样道理，在字节串上是没有普通字符串那样的格式化操作的。

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

如果想在字节串上做任何形式的格式化操作，应该使用普通的文本字符串然后再做编码。示例如下：

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME              100     490.10'
>>>
```

最后，需要注意的是使用字节串会改变某些特定操作的语义——尤其是那些与文件系统相关的操作。例如，如果提供一个以字节而不是文本字符串来编码的文件名，文件系统通常都会禁止对文件名的编码/解码。示例如下：

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('spicy')
...

>>> # Get a directory listing
>>> import os
>>> os.listdir('.')          # Text string (names are decoded)
['Jalapeño.txt']
>>> os.listdir(b'.')        # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

请注意这个例子中的最后部分，本例中以字节串作为目录名从而导致产生的名称以未经编码的原始字节形式返回。在显示目录内容时，文件名包含了原始的 UTF-8 编码。有关文件名的处理请参阅 5.15 节。

最后要说的是，有些程序员可能会因为性能上有可能得到提升而倾向于将字节串作为文本字符串的替代来使用。尽管操纵字节确实要比文本来的略微高效一些（由于同 Unicode 相关的固有开销较高），但这么做通常会导致非常混乱和不符合语言习惯的代码。我们常会发现字节串和 Python 中许多其他部分并不能很好地相容，这样为了保证结果的正确性，我们只能手动去执行各种各样的编码/解码操作。坦白地说，如果要同文本打交道，在程序中使用普通的文本字符串就好，不要用字节串。